Übungen zu

Middleware

Wintersemester 2003/04

Teil III: JINI

A.1 Überblick über JINI

- Java Intelligent Network Interface (JINI)
- Von Sun 1998 entwickelter Standard zur Kommunikation von "intelligenten" Geräten
- Basiert auf Java und JavaRMI
- Standard-Interfaces in `net.jini.*`
- Referenzimplementierung von Sun in `com.sun.jini.*`
- Aktuelle Version 2.0 (Juni 2003)

A Überblick über die 9. te Übung

- Einführung
- Discovery
- Lookup-Service
- Aufbau eines JINI Services
- Leasing

A.1 Konzepte

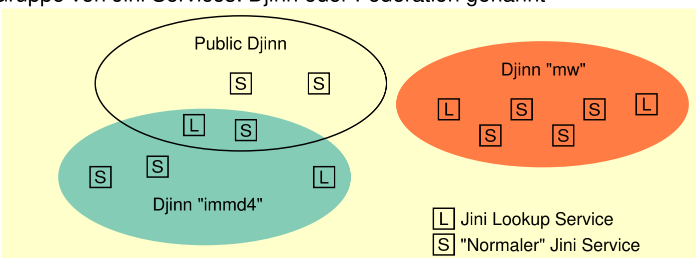
- Discovery
- Lookup
- Leasing
- Remote Events
- Distributed Transactions
- Security
- RMI Versionen

A.1 Dokumentation

- Jan Newmarch: A Programmer's Guide to Jini Technology, APress, 2000
  http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml
- Spezifikationen zu Jini
- Informationen zu Klassen im JavaDoc-Format
- Quellen
- Komplettes Paket (Version 2.0.001) in **`/local/java-lib/jini`**

A.1 Architekturüberblick

- Beteiligte Komponenten:
  - ◆ Service
  - ◆ Nutzer von Servicen
  - ◆ mindestens ein Lookup-Service
- Gruppe von Jini Services: Djinn oder Federation genannt



- Flacher Namensraum der Djinns

A.2 Discovery

- Wie findet man einen Lookup Services in einem Djinn?
- Wie spricht man mit dem Lookup Service?
- Lösungen
  - ◆ Multicast Discovery
    - Multicast Request Protocol
    - Multicast Announcement Protocol
  - ◆ Unicast Discovery Protocol
  - ◆ Join Protocol
    - Legt fest, wie ein Service die Discovery-Mechanismen verwenden soll
    - z.B. zufällige Zeit vor Discovery warten

1 Multicast Announcement Protocol

- Lookup Services senden regelmäßig Multicast Announcement Pakete
  - ◆ IP-Adresse und Port des Lookup Services
  - ◆ Lookup Service ID
  - ◆ Namen der Djinns, zu denen der Lookup Service gehört
- Spezielle Multicast Gruppe
  - ◆ Addresse: 224.0.1.84 **(`JINI-ANNOUNCEMENT.MCAST.NET`)**
- Informationen für Unicast Discovery im Multicast Announcement Paket

2 Multicast Request Protocol

- Neuer Jini Service sendet Multicast Request Pakete mit:
  - Eigenem TCP-Port (in Protokollversion 2 auch die IP-Adresse)
  - Namen der Djinns, die ihn interessieren
  - Liste schon bekannter Lookup Services
- Spezielle Multicast Gruppe
  - Addresse: 224.0.1.85 **`(JINI-REQUEST.MCAST.NET)`**
- Lookup Services antworten mit Informationen für Unicast Discovery
  - Eigene IP-Adresse und Port
  - Namen der Djinns, zu denen der Lookup Service gehört
- Multicast Requests werden periodisch wiederholt (alle 5 sec.)

3 Unicast Discovery Protocol

- Man kennt den *Contact Point* des Lookup Service
  - durch Multicast Discovery
  - durch vorkonfigurierte Informationen, z.B. URL **`jini://host:port/`**
- Einfaches Request Paket
  - nur Protokollversion
- Standard TCP-Port: 4160
- Reply Paket
  - Namen der Djinns, zu denen der Lookup Service gehört
  - Proxy für Lookup Service als **`java.rmi.MarshalledObject`** in **`java.io.ObjectOutputStream`**
- Proxy implementiert Standard Interface für Lookup Service **`net.jini.core.lookup.ServiceRegistrar`**

4 Implementierung

- Alle Klassen für Discovery Protokolle in **`net.jini.discovery`**
- für Unicast Discovery:
  - LookupLocator
- für Multicast Discovery:
  - LookupDiscovery
- Ausbreitung der Multicast Pakete wird über Property **`net.jini.discovery.ttl`** gesteuert
  - Time-To-Live (TTL)
  - IP-Router zählt TTL bei jedem Paket runter
  - bei Multicast administrative Grenzen der TTL (<8 lokales Netz, <16 innerhalb der Uni, <64 innerhalb Deutschlands)

4 Implementierung (2)

- Unicast Discovery mittels LookupLocator:

```
public class LookupLocator {
    public LookupLocator(java.lang.String url)
                throws java.net.MalformedURLException ...
    public LookupLocator(java.lang.String host,int port) ...

    public ServiceRegistrar getRegistrar()
            throws IOException, ClassNotFoundException ...
    public ServiceRegistrar getRegistrar(int timeout)
            throws IOException, ClassNotFoundException ...

    public String getHost() ...
    public int getPort() ...
}
```

4 Implementierung (3)

- Beispiel:

```
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RMISecurityManager;

...

LookupLocator lookup = null;
ServiceRegistrar registrar = null;

System.setSecurityManager(new RMISecurityManager());

try {
    lookup = new LookupLocator("jini://faui40c");
    registrar = lookup.getRegistrar();
} catch (Exception e) { ... }

System.out.println("Registrar found");

...
```


4 Implementierung (4)

- Multicast Discovery mittels LookupDiscovery

```
public class LookupDiscovery {
    public LookupDiscovery(String[] groups) ...
    public LookupDiscovery(String[] groups,
                           Configuration config) ...
    public void setGroups(String[] newGroups) ...
    public String[] getGroups() ...
    public void addGroups(String[] newGroups) ...
    public void removeGroups(String[] oldGroups) ...
    public static final String[] ALL_GROUPS = ...
    public static final String[] NO_GROUPS = ...

    public void addDiscoveryListener(DiscoveryListener l) ...
    public void removeDiscoveryListener(DiscoveryListener l)

    public ServiceRegistrar[] getRegistrars() ...
    public void discard(ServiceRegistrar reg) ...

    public void terminate() ...
}
```


4 Implementierung (5)

- Das Interface DiscoveryListener (Interface eines Suchenden)

```
public interface DiscoveryListener {
    public void discovered(DiscoveryEvent e);
    public void discarded(DiscoveryEvent e);
}
```

  - **`discarded()`** wird nur aufgerufen, wenn die Applikation **`discard()`** an LookupDiscovery aufruft (z.B. wegen eines Fehlers bei der Kommunikation)
  - In der *Jini Discovery Utilities Specification* wird festgelegt, dass gefundene Lookup Services gepuffert werden bis ein DiscoveryListener registriert wurde.

- Parameter vom Typ DiscoveryEvent

```
public class DiscoveryEvent extends java.util.EventObject {
    public Map getGroups() ...
    public ServiceRegistrar[] getRegistrars() ...
}
```

  - wird an alle registrierten DiscoveryListener verteilt

5 Implementierung (6)

- Beispiel:

```
import net.jini.discovery.*
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RMISecurityManager;

public class MulticastRegister implements DiscoveryListener
{
    public MulticastRegister() {
        System.setSecurityManager(new RMISecurityManager());
         LookupDiscovery discover = null;
         try {
             discover =
             new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
         } catch(Exception e) {...}
         discover.addDiscoveryListener(this);
     }
    public void discovered(DiscoveryEvent evt) {
        ServiceRegistrar[] registrars = evt.getRegistrars();

        if (registrars != null && registrars[0] != null)
            System.out.println("found a service locator");
    }
    public void discarded(DiscoveryEvent evt) { }
}
```

A.3 Lookup Service

- Zum Auffinden von Services
- In der Spezifikation festgelegtes Interface
  - im Package **`net.jini.core.lookup`**
- Interface, das von den Proxies aller Lookup Services implementiert wird
  - **`net.jini.core.lookup.ServiceRegistrar`**
- Beliebige Implementierungen
- Referenzimplementierung von Sun
  - in **`com.sun.jini.reggie`**

1 ServiceRegistrar – Lookup

- Methoden zum Suchen eines Service über **`ServiceTemplate`**

```
import net.jini.core.lookup.*;
public interface ServiceRegistrar {
    public java.lang.Object lookup(ServiceTemplate tmpl)
                            throws java.rmi.RemoteException;

    public ServiceMatches lookup(ServiceTemplate tmpl,
                                 int maxMatches)
                            throws java.rmi.RemoteException;
    ...
}
```

- **`lookup`** kann eine Liste gefundener Proxys für den Service liefern
- **`maxMatches`** ist dann die maximale Anzahl an Servicen, die gefunden werden sollen
- Exception z.B. wenn der Service nicht deserialisiert werden kann.

2 ServiceRegistrar – Lookup (2)

- **`ServiceMatches`**: Sammlung von **`ServiceItems`**

```
public class ServiceMatches {
    public ServiceItem[] items;
    public int totalMatches;
}
```

  - Achtung: **`items.length`** ≤ **`totalMatches`**
- **`ServiceItem`**: Informationen über den gefundenen Service

```
public class ServiceItem {
    public Entry[] attributeSets;
    public Object service;
    public ServiceIDserviceID ;
}
```

(siehe Registrierung von Services)

3 ServiceTemplate

- Angaben zum gesuchten Service

```
public class ServiceTemplate {
    public java.lang.Class[] serviceTypes;
    public Entry[] attributeSetTemplates;
    public ServiceID serviceID;
}
```

- **`serviceTypes`** beschreibt die geforderten Service-Interfaces (als Menge von **`Class`**-Objekten von Service-Interfaces)
- mittels **`attributeSetTemplates`** können zusätzliche Attribute angegeben werden.
- **`ServiceID`** (meist **`null`**, da unbekannt)
- Alle Felder können auch **`null`** sein und werden dann nicht für den Vergleich benutzt

4 Ablauf beim Finden eines Service

- **Class**-Objekt der gewünschten Interfaces erzeugen und in ein Service Template eintragen
- Entries erzeugen, teilweise ausfüllen und in **ServiceTemplate** eintragen
- **ServiceID** in **ServiceTemplate** leer lassen
- Mit **LookupDiscovery** Proxies (**ServiceRegistrar**) für Lookup Services suchen
  - Wenn neuer **ServiceRegistrar**, dann dort mit **ServiceTemplate** suchen
- Mehrere Treffer werden als **ServiceItems** (mit Proxies) geliefert
- Achtung: Proxy funktioniert meist nur zeitlich begrenzt (siehe Leasing)

5 ServiceItem

- Beschreibung der Services durch ein **ServiceItem**

```
public class ServiceItem {
    public Object service;
    public Entry[] attributeSets;
    public ServiceIDserviceID ;
}
```

- Proxy-Objekt für Service (muss serialisierbar sein)
- Menge von **Entry**-Objekten als Attribute
- ServiceID
  - wird beim Registrieren neu vergeben, wenn **null**
  - weltweit eindeutig

5 ServiceRegistrar – Registrierung

- Methode zum Registrieren eines Service über **ServiceItem**

```
public interface ServiceRegistrar {
    public ServiceRegistration register(ServiceItem item,
                                        long leaseDuration)
                            throws java.rmi.RemoteException;
    ...
}
```

- Registrierung ist temporär (siehe Leasing)

5 Entry

- Attribut eines Services: abgeleitet von **net.jini.core.entry.Entry**
- alle Felder müssen öffentlich sein
- jedes Feld wird separat serialisiert
- Ein Standardkonstruktor (= ohne Argumente) muss vorhanden sein
- Einige vordefinierten Entries in **net.jini.lookup.entry**:
  - **Address**, **Comment**, **Location**, **Name**, **ServiceInfo**, **ServiceType**, **Status**

5 ServiceRegistration

- Beeinflussung der Registrierung

```
public class ServiceRegistration {
    ServiceID getServiceID();
    void addAttributes(Entry[] attrSets);
    void modifyAttributes(Entry[] attrSetTemplates,
                          Entry[] attrSets);
    void setAttributes(Entry[] attrSets);
    Lease getLease()
}
```

- Abfragen der ServiceID und nachträgliches Ändern der Attribute

6 Ablauf beim Registrieren eines Service

- Eigenen Proxy erzeugen und in `ServiceItem` eintragen
- Entries erzeugen, ausfüllen und in `ServiceItem` eintragen
- Mit `LookupDiscovery` Proxies für Lookup Services suchen
  - Wenn neuer `ServiceRegistrar`, dann dort mit `ServiceItem` anmelden
  - Wenn noch keine ServiceID, dann ServiceID aus aktueller Anmeldung in `ServiceItem` eintragen, sonst beibehalten
- Weiter nach Lookup Services suchen und dort anmelden
- Achtung: Anmeldung zeitlich begrenzt (siehe Leasing)

A.4 Aufbau von Jini Services

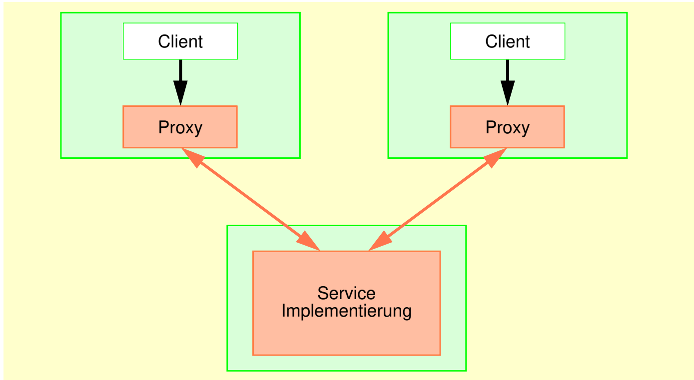
- Implementieren ein oder mehrere Interfaces
- meist: Interfaces von `java.rmi.Remote` abgeleitet

1 Proxies

- Benutzer eines Jini Service brauchen lokalen Proxy
- Proxy-Objekt vom Service erzeugt und beim Lookup Service registriert
- "Fragmentiertes Objektmodell"
  - Beliebige Verteilung der Funktionalität
  - Aber alle Proxies sind gleich

1 Proxies

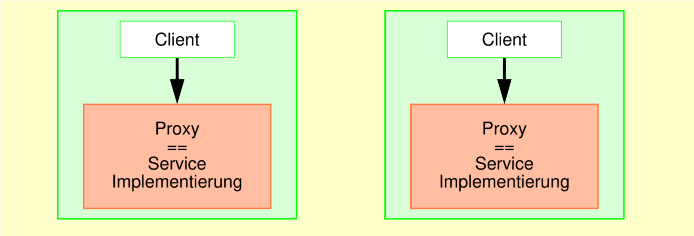
- Fall 1: Proxy ist Stub



  - Kommunikation über JavaRMI, CORBA, Sockets, ...
  - z.B. RMI-Stubs ohne Änderung als Proxies verwendbar

1 Proxies

- Fall 2: Proxy macht alles lokal



- Beliebige Abstufungen dazwischen

A.5 Leasing

- Neue Fehlerklassen bei verteilten Objekten:
  - Netzwerkausfall
  - Ausfall eines Services
- Alle Proxies, Registrierungen, usw. sind daher nur zeitlich begrenzt gültig
  - ◆ Ausnahme Proxies für Lookup Services

1 Lease

- Festgelegtes Interface in **`net.jini.core.lease`**

```
public interface Lease {
    void cancel();
    void renew(long duration);
    long getExpiration();
    ...
}
```

- Hat begrenzte Dauer, die vom Lease-Geber festgelegt wird
  Nach Ablauf ist die Ressource nicht mehr benutzbar
- Lease vor Ablauf zurückgeben: **`cancel`**
- Verlängerung beantragen: **`renew`**
  muss vom Lease-Geber bewilligt werden
  - Spezielle Konstanten: **`Lease.ANY`**, **`Lease.FOREVER`**
- Restlaufzeit (in absoluter Zeit) abfragen:
  `lease.getExpiration() - System.currentTimeMillis();`

2 Lease Verlängerung

- Normalerweise: Lease solange verlängern, bis nicht mehr benötigt
- **`com.sun.jini.lease.LeaseRenewalManager`**

```
public class LeaseRenewalManager {
    void remove (Lease lease);
    void cancel (Lease lease);
    void renewFor (Lease lease, long desiredDuration,
                              LeaseListener listener);
    void renewUntil (Lease lease, long desired Expiration,
                              LesaeListener listener);
    void setExpiration(Lease lease, long expiration);
    long getExpiration (Lease lease);
    ...
}
```

- ◆ verwaltet beliebige Leases
- ◆ informiert per Callback, wenn eine Lease nicht verlängert werden konnte

3 Lease Implementierung

- Sollte jeder Proxy haben
- Meist eigene Strategie
- Basisklassen mit Grundfunktionalität in **`com.sun.jini.lease`** und **`com.sun.jini.lease.landlord`**

A.6 sonstiges

- SecurityManager braucht eine Policy
- Policy-Datei: **`start.policy`**

```
grant {
    permission java.security.AllPermission;
};
```

- Anwendung starten

```
java -Djava.security.policy=start.policy <Klassenname>
```

- LookupService reggie starten

```
/proj/i4mw/pub/aufgabe6/reggie.sh
```

◆ (verwendet einen Web-Server auf faui40b:8080)